2INC0 - Operating Systems

Processes, Threads and Scheduling

Geoffrey Nelissen

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e
Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lectures 2 and 3)
- **Concurrency and synchronization**
  - atomicity and interference (lecture 4)
  - action synchronization (lecture 5)
  - condition synchronization (lecture 6)
  - deadlock (lecture 7)
- **File systems** (lectures 8)
- **Memory management** (lectures 9 and 10)
- **Input/output** (lecture11)

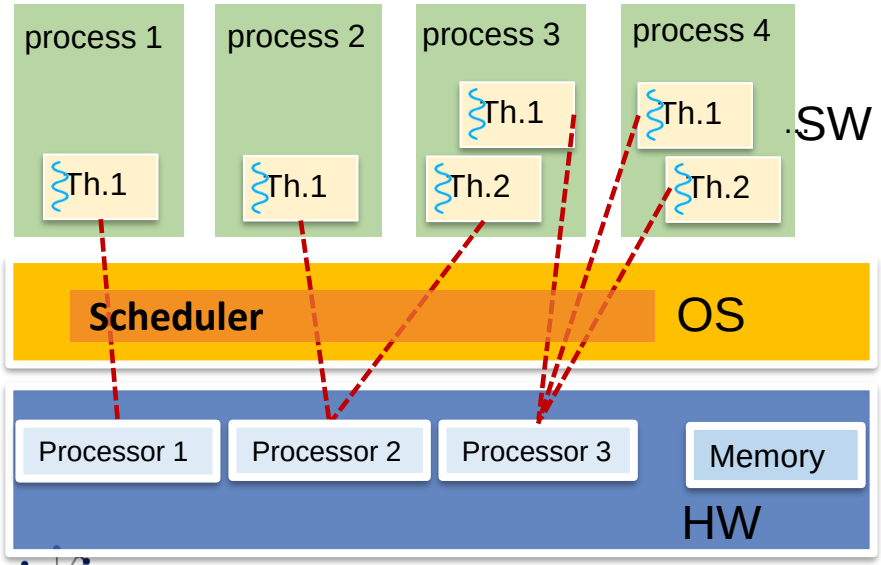
Ingredients of concurrent execution: processes and threads

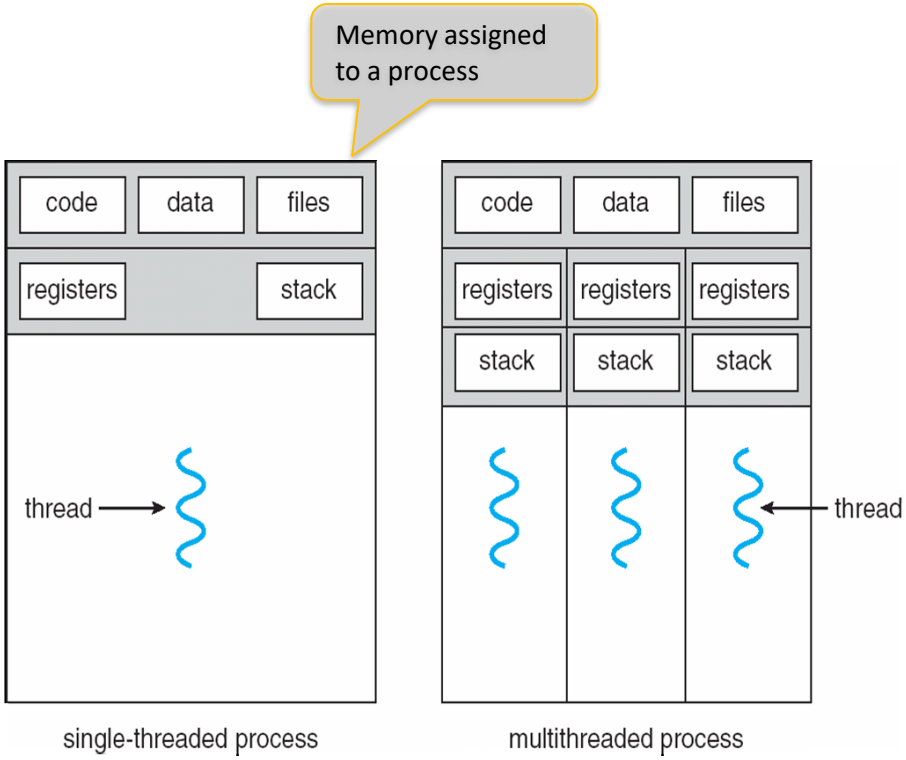
Process

- It is a **program in execution**.
- It has a context of execution
- A process owns resources (has memory and can own other resources such as files, etc.)

Thread

- A **dispatchable unit of work within a process**
- Threads within a process share code and data segments (i.e., share memory address space)





Agenda

- Processes
- Threads
- Scheduling

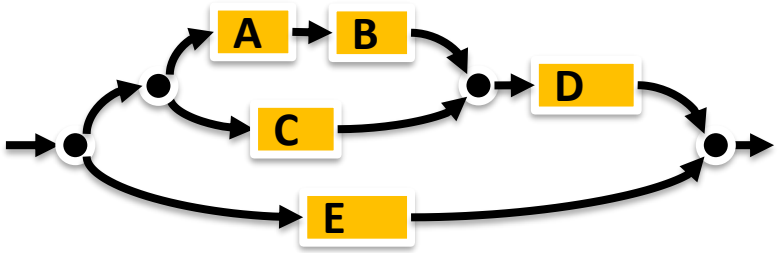
Exercise (5 minutes): process creation and synchronization

```c
#include <stdio.h>
#include <sys/wait.h> // wait()
#include <unistd.h> // fork()

void A() { printf("A completed \n"); }
void B() { printf("B completed \n"); }
void C() { printf("C completed \n"); }
void D() { printf("D completed \n"); }
void E() { printf("E completed \n"); }
```

```c
int main()
{
    int status;
    pid_t ...
    .
    .
    .
    return 0;
}
```

Write a pseudo-code to run functions A, B, C, D, and E on three concurrent processes such that the execution of these functions follows the diagram below (arrows show precedence constraints):



- C can run concurrently with A, B, and E but must be completed before D.
- B must be executed after A and before D.

Hint: to wait for a certain process to complete, use

`waitpid(pid, &status, 0);`

pid (should have a pid_t type) and is the ID of the process to wait for.

To fork a new process:

`pid_t pid = fork();`

If pid == 0, it is the child process, otherwise the parent

- Build the solution together
- Use only three processes max

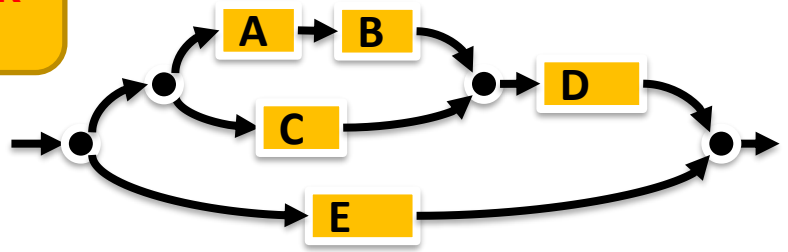
Exercise

!!! A fully correct solution would require to **check for error codes**

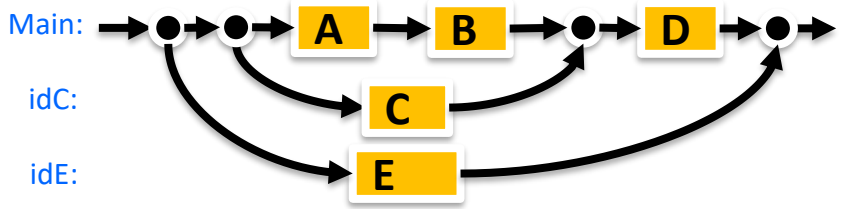
```
#include <stdio.h>
#include <sys/wait.h> // wait()
#include <unistd.h> // fork()

void A() { printf("A completed \n"); }
void B() { printf("B completed \n"); }
void C() { printf("C completed \n"); }
void D() { printf("D completed \n"); }
void E() { printf("E completed \n"); }

int main()
{
    int status;
    pid_t idC, idE;
    idE = fork();
    if (idE > 0)
    {
        idC = fork();
        if (idC > 0)
        {
            A();
            B();
            waitpid(idC, &status, 0); // for idC to join
            D();
            wait(NULL);
            printf("workload processed.\n");
        }
        else
        {   C();   }
     }
     else
     {   E();   }
    return 0;
}
```



- C can run concurrently with A, B, and E but must be completed before D.
- B must be executed after A and before D.



A quick peek at the next lecture on concurrency

TU/e

What are the outputs that we may see on the screen (all of them)?

Hello Child
Bye
Hello Parent
Child Terminated
Bye

or

Hello Parent
Hello Child
Bye
Child Terminated
Bye

or

Hello Child
Hello Parent
Bye
Child Terminated
Bye

```c
// C program to demonstrate working of wait()
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>

int main()
{
    if (fork()== 0)
        printf("Hello Child\n");
    else
    {
        printf("Hello Parent\n");
        wait(NULL);
        printf("Child Terminated\n");
    }

    printf("Bye\n");
    return 0;
}
```

IRiS

Source: https://www.geeksforgeeks.org/wait-system-call-c/

Reminder

Remember: the content of the preparatory material is also part of the exam
(e.g., process state, inter-process communication, etc.)

Agenda

- Processes
- Threads
- Scheduling

Process and concurrency

- Using processes allows to use the platform **parallelism**:
  - Example: if the processor has more than one core, two or more different process can make **simultaneous progress**

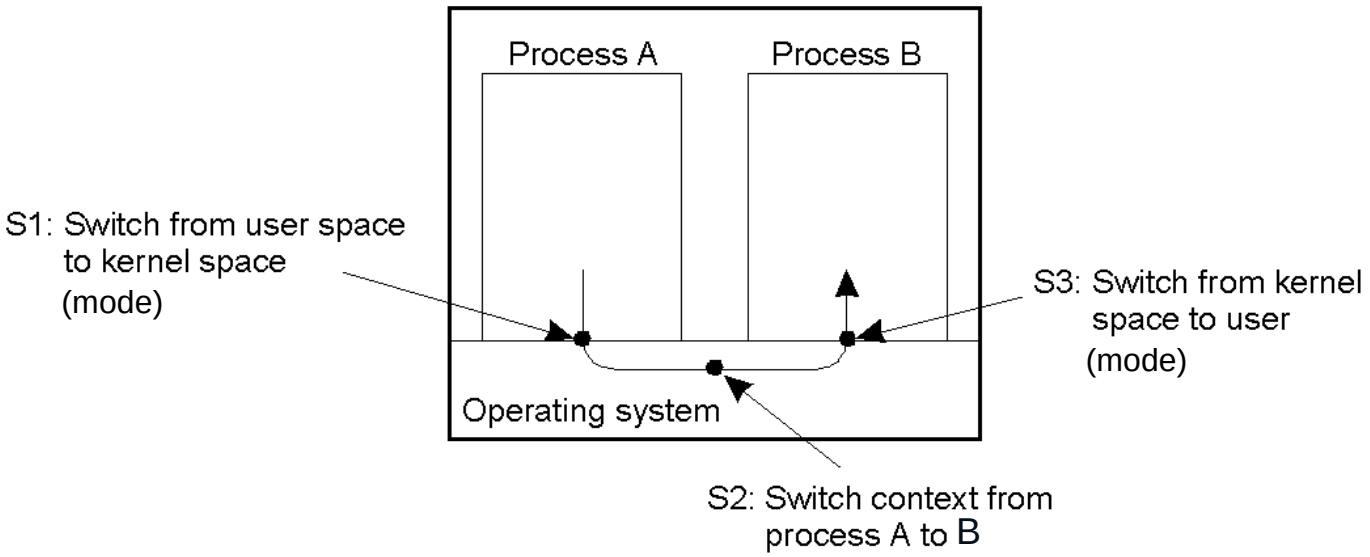
- It also allows for **concurrency**:
  - The same resource (e.g., a core or network driver) can be **time shared between processes**
    ➔ **improve responsiveness**

  - If a process is **waiting** (e.g., to get access to a resource, or to receive data from another process), then the **kernel** can **schedule another process**
    ➔ **more efficient use of the system resources**

Process: disadvantages

- **Creation overhead** (e.g., memory space creation/copy)
- **Complex inter-process communication**
- **Process switching overhead**: ***mode + context switch***
  - **save/restore context of execution**

Each (synchronized) *inter-process communication (IPC)* on a single-processor system may generate several mode and context switches



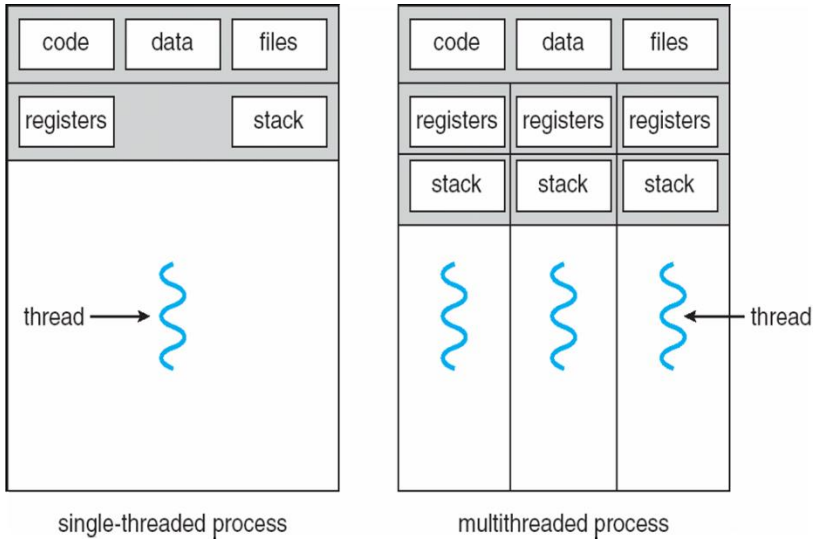
Process vs thread

Process

- defines a memory space
- defines ownership on other resources
- has at least one associated thread of execution
- Context of execution saved in **a PCB**

Thread

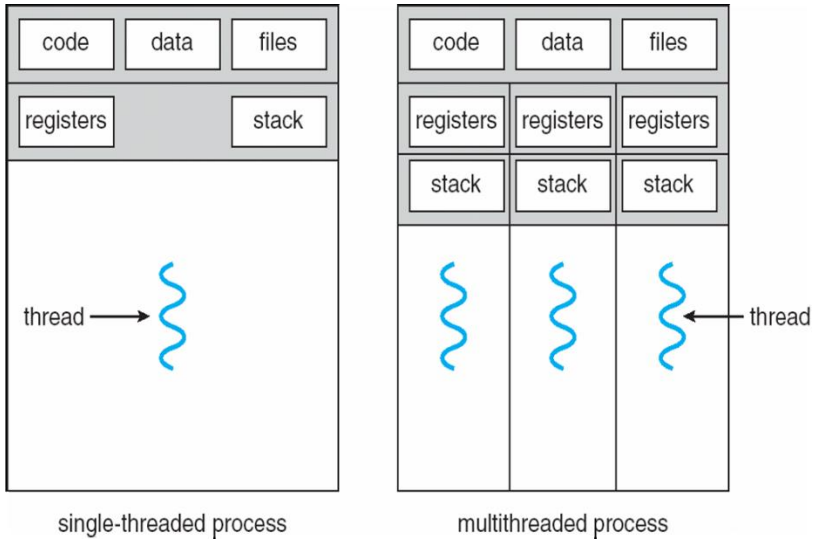
- all threads of a process can operate in their process' address space
  - several threads may share access to the same variables, code or files
- has an associated execution state (PCB is extended with thread info)
  - program counter, stack counter, return addresses of function calls, values of processor registers



Process vs thread

Threads operate faster than processes due to the following reasons

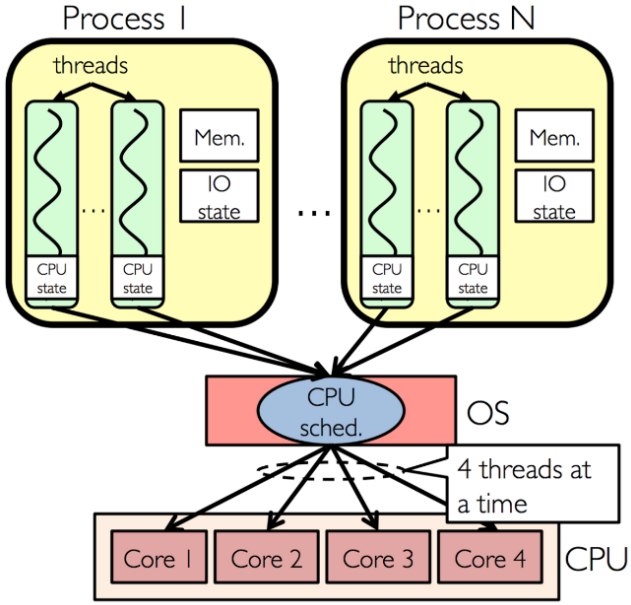
- **Thread creation and termination is much faster (no memory address space copy needed)**
- **Context switching between threads of the same process is much faster** (no need to switch the whole address space, **only swap cpu registers content**)
- **Communication between threads is faster and more at the programmer's hand than between processes (shared address space)**



Reasons to introduce threads

- **Increase concurrency level** (less costly than by using processes)
  - Better performance
    - hide latency (while waiting CPU can do something useful in another thread)
    - increase responsiveness (divide work)
    - exploit platform parallelism (multiple processors → multiple threads)

- **Use "natural" concurrency within a program**
  - natural organization, structure, e.g.
    - one thread per event
    - one thread per task
    - one thread per resource
    - one thread to handle each user interface
    - …

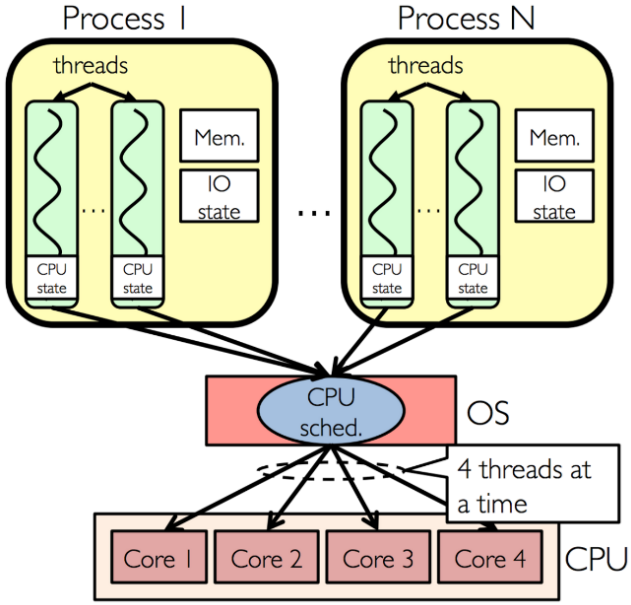


What is the downside of using threads?

Reasons to introduce threads

- **Increase concurrency level** (less costly than by using processes)
  - Better performance
    - hide latency (while waiting CPU can do something useful in another thread)
    - increase responsiveness (divide work)
    - exploit platform parallelism (multiple processors → multiple threads)

- **Use "natural" concurrency within a program**
  - natural organization, structure, e.g.
    - one thread per event
    - one thread per task
    - one thread per resource
    - one thread to handle each user interface



Downside: **no protection against other threads in the same process** (faults, memory sharing)

Agenda

- Processes
- Threads
  - Using threads with POSIX
  - Managing threads in an Operating System
- Scheduling

Thread creation and merging (POSIX)
pthread_create() and pthread_join()

TU/e

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep().
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Print something from the thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

- **pthread_create(.) creates a new thread**
  - The first argument is a pointer to thread_id which is set by this function
  - The second argument specifies attributes. If the value is NULL, then default attributes shall be used. For example, stack size is an attribute that can be set.
    - See here to learn about attributes: https://linux.die.net/man/3/pthread_attr_init
  - The third argument is the name of the function to be executed for the thread to be created.
  - The fourth argument is used to pass arguments to the function.
- **pthread_join(.) is the equivalent of waitpid() for processes**
  - A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.
- **pthread_detach(.) will detach the thread** ➔ resources held by the thread will be automatically released when the thread completes without the need for another thread to call pthread_join(.)
- **pthread_cancel(.) stops a thread**

```
status = pthread_create (&thread_id, attr, func, arg_of_func);
status = pthread_join (thread_id, &result);
```

IRiS

Source: https://www.geeksforgeeks.org/multithreading-c-2/

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Let us create a global variable
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Change global variables
    ++g;
}

int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads
    for (i = 0; i < 8; i++) {
        pthread_create(&tid, NULL, myThreadFun, (void *)&tid);
        pthread_join((&tid, NULL);
    }

    return 0;
}
```

What is the final value of g when we reach to "return 0"?

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Let us create a global variable
int g = 0;

// The function to be executed by all threads
void *myThreadFun(void *vargp)
{
    // Change global variables
    ++g;
}

int main()
{
    int i;
    pthread_t tid;

    // Let us create three threads
    for (i = 0; i < 8; i++) {
        pthread_create(&tid, NULL, myThreadFun, (void *)&tid);
        pthread_join((&tid, NULL);
    }

    return 0;
}
```

What is the final value of g when we reach to "return 0"?

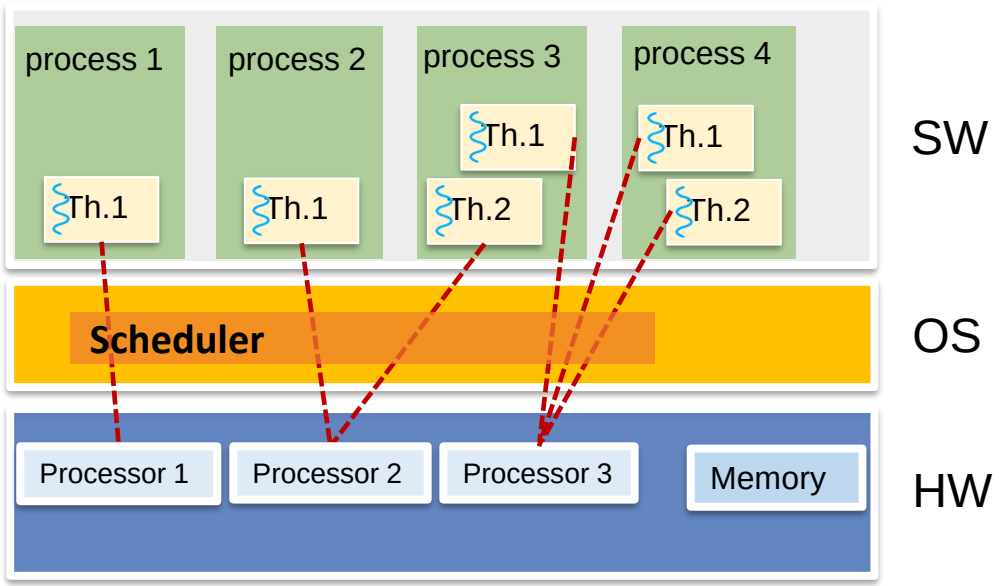
8

Agenda

- Processes
- Threads
  - Using threads with POSIX
  - Managing threads in an Operating System
- Scheduling

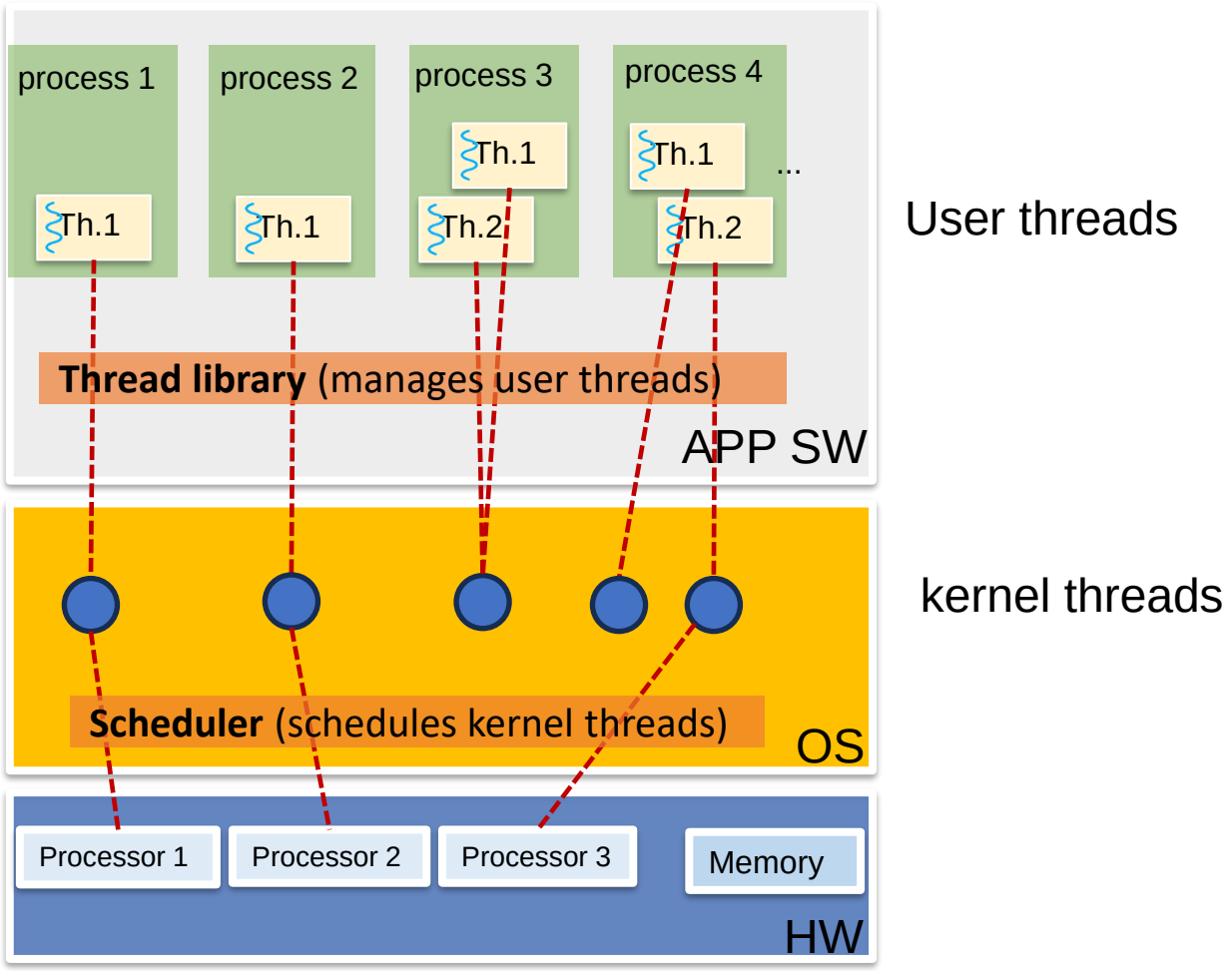
User and kernel threads

Simplified view with no differentiation between user and kernel threads



User and kernel threads

What may actually be happening:



User and kernel threads

- Threads are supported either at user level or by kernel
  - user threads
  - kernel threads

- ***User threads*** supported above the kernel (no kernel support)
  - **via thread libraries**
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

- ***Kernel threads*** supported by all modern OS (**kernel support**)
  - Windows
  - Solaris
  - Linux, Unix (some versions), Android
  - Mac OS X

Thread libraries can also be used on top of OS's supporting kernel threads.

Needed: A mapping model between user threads and kernel threads.

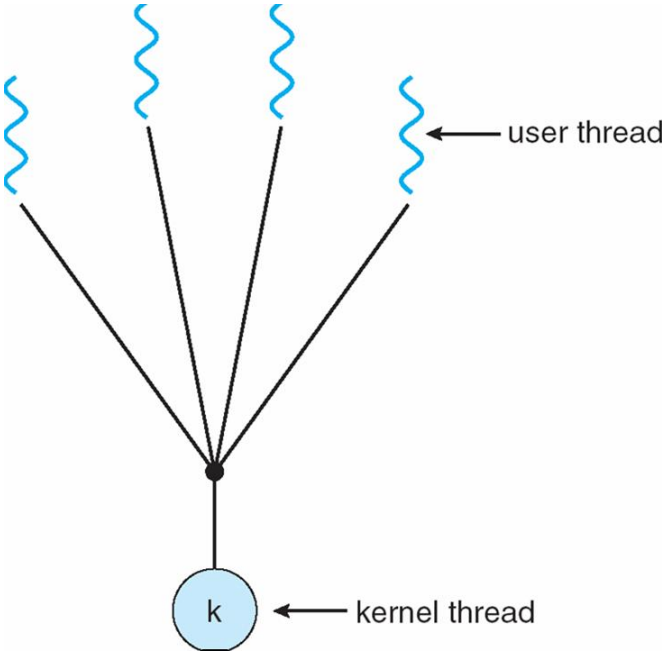
Multithreading mapping models

- **Many-to-one**
- One-to-one
- Many-to-many
- Two-level model;

The following slides are provided to help you study. The reference book is the best source of information on that topic

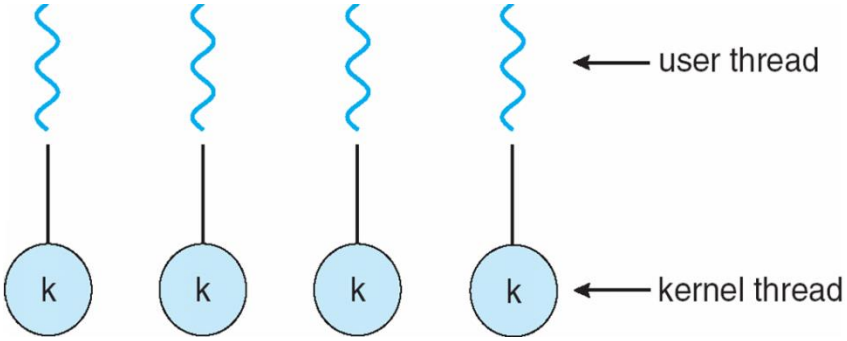
Many-to-one model

- User threads mapped to single kernel thread
- ***Advantage**:* Thread management by thread lib
  - ***no*** need for ***kernel involvement***,
    ➔ **fast**
    ➔ **easy to deploy**
- ***Disadvantage**:* Only one user thread can access the kernel at a given time
  - multiple threads ***cannot run in parallel*** on different processors
  - a ***blocking system call*** from one user thread blocks all user threads of the process



One-to-one model

- Each user-level thread maps to a kernel thread



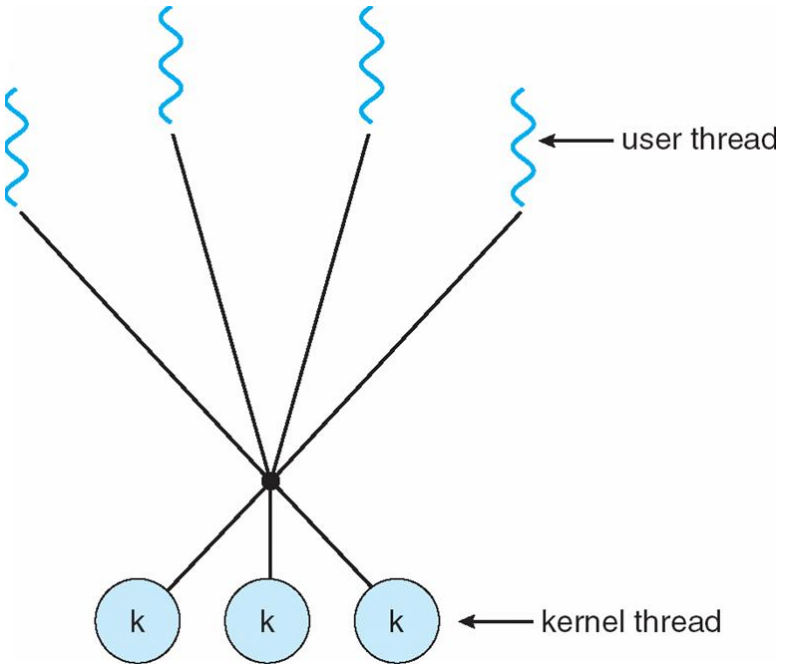
- ***Advantage****:* allows for **concurrency** between all threads
- ***Disadvantage****:* the kernel must manage all threads (state, schedule, …)
  - **Impacted performance in case of many user threads**
- Examples: Windows, Linux, Android

Many-to-many model

- Allows OS to create a (limited) number of kernel threads
- #kernel_threads ≤ #user_threads

- ***Advantages***:
  - **concurrency**
  - **bounded performance cost** for the kernel
- ***Disadvantage***: more **complex** to implement

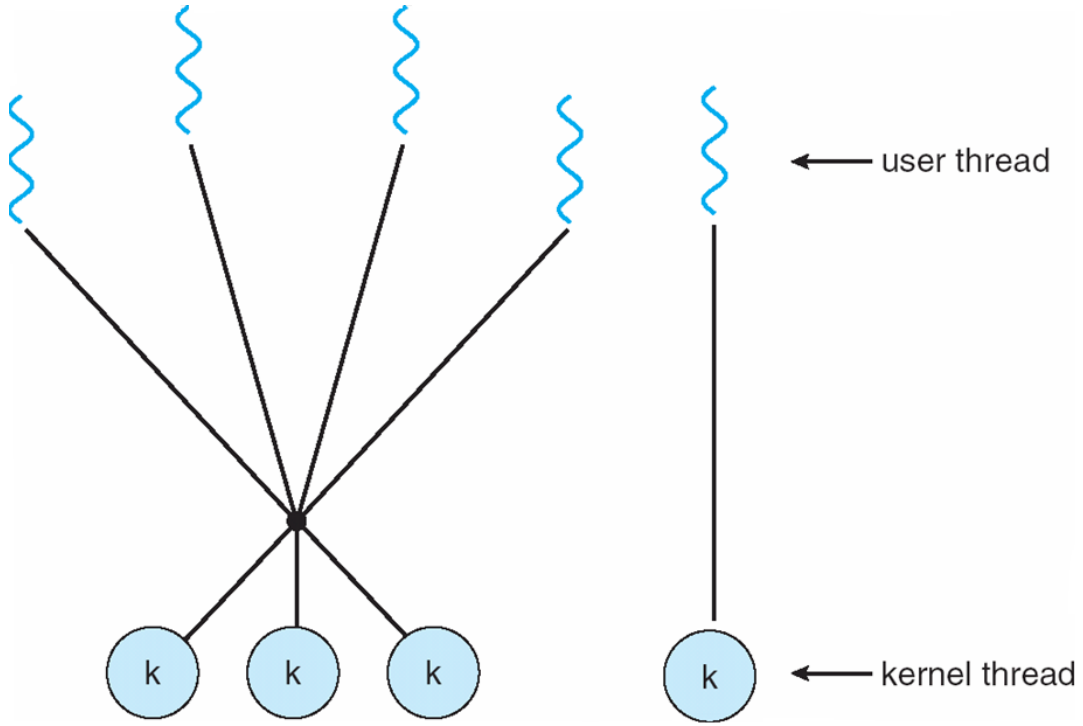
- Examples
  - Windows NT/2000 with the *ThreadFiber* package
  - Solaris 8 and earlier



Concurrency level is limited by the number of ***kernel threads***.

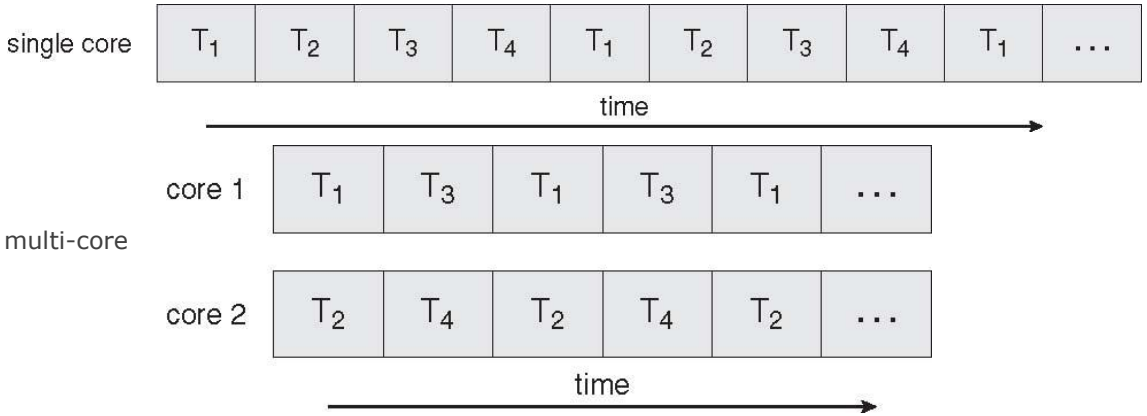
Two-level model

- **Similar to many-to-many**, except that it **allows a user thread to be bound to a kernel thread**



Thread switching

- Threads are executed in an interleaved way



- Switching between threads: two possibilities
  - switching as a **kernel activity**
    - kernel maintains execution state of thread
    - similar to process switching
  - switching handled by **user-level thread library**
    - the library maintains execution state of threads (kernel knows kernel threads only)
    - the library must obtain control in order to switch threads
    - responsibility of **programmer** to **call the library to yield execution** – **'cooperative multithreading'**

Thread pools

- **Not efficient to often create and delete many threads dynamically**.
- **Alternative**: given an application, create a number of threads in a pool where they wait for work.
- Advantages:
  - Usually ***slightly faster to service a request*** with an existing thread than creating a new thread
  - Allows the ***number of user threads*** in the application(s) to be ***bounded by the size of the pool***
- Example
  - ThreadPoolExecutor (Java): assigns app tasks to a thread pool

Agenda

- Processes
- Threads
- **Scheduling**

Scheduling

- **Problem:**
  Only one task (e.g., process, thread, I/O request, memory page) can access a hardware resource at a time.

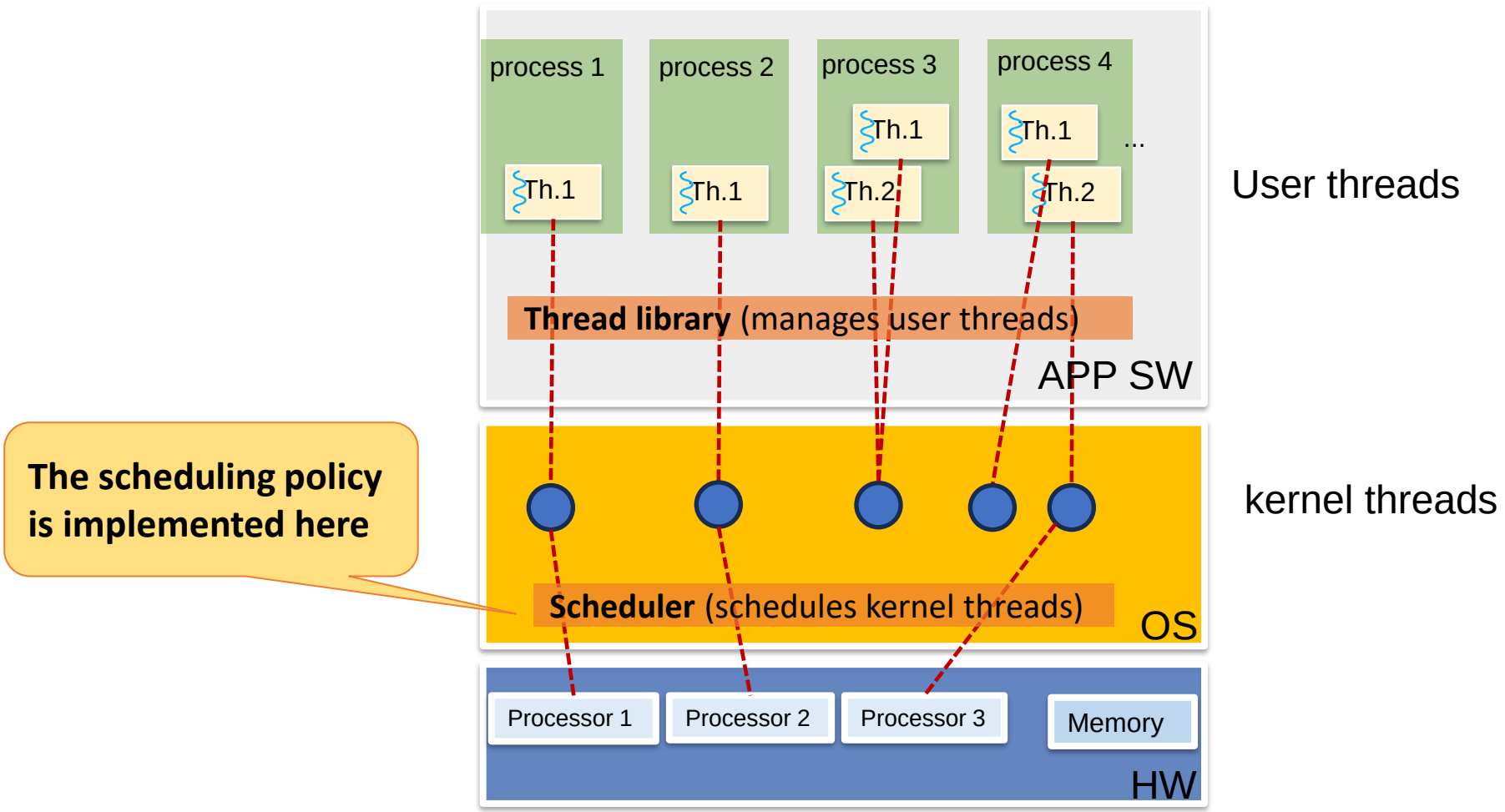
- **Solution:**
  Scheduling, i.e., decide **what** (executes/is loaded/has access to) **where** and **when**

Covered in the next slides

- **Examples:**
  - **Processor scheduling:** decide which **task** (thread/process) executes **on** each **core** at every given time
  - **Memory scheduling:** decide which **memory page of a process** is **loaded in** each **page frame of** the **main memory** at every given time

Covered in the lectures on memory management

Processor scheduling



Common processor scheduling algorithms

- **First-Come First-Served (FCFS)**
- **Shortest-Job-First (SJF)**
- **Round Robin (RR)**
- **Priority scheduling**
  - Rate Monotonic (RM)
  - Deadline Monotonic (DM)
  - Earliest Deadline First (EDF)
- **Multilevel queue**
- **Multilevel feedback queue**

Terminology

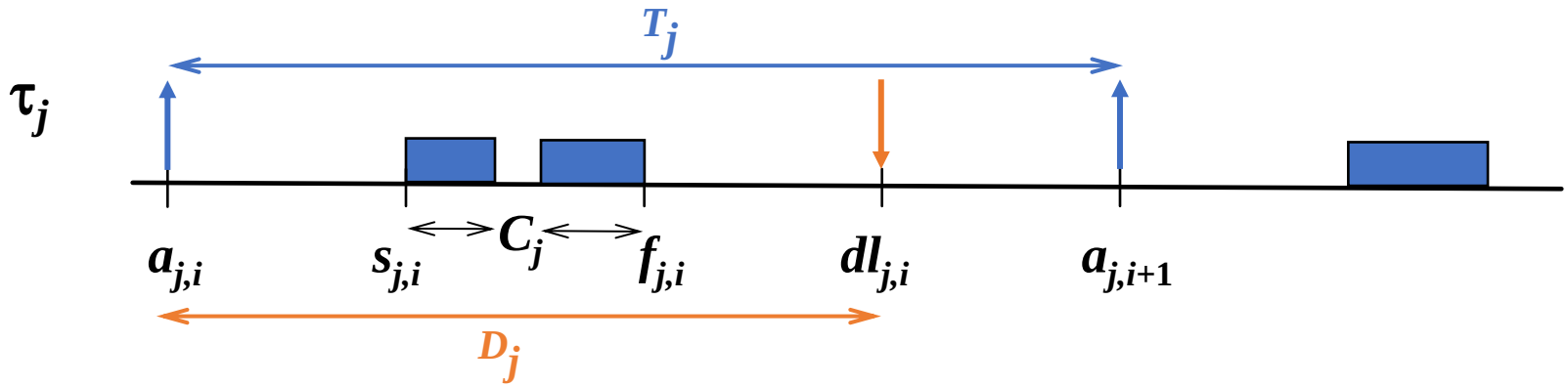
- Decision mode defines when scheduling decisions are taken
  - Preemptive vs non-preemptive
  - Time-based vs event-based

- Priority function defines what ready task to choose for execution
- Arbitration rule breaks ties

- Waiting time = time spent in the ready queue
- Response time = time until first response
- Turnaround time = time to completion
- Throughput = number of jobs completed per time unit

Beware: in most of the scientific literature (and other courses), *response time* is defined as the time elapsing from arrival to completion. In our textbook, this is called *turnaround time.* To avoid confusion, I will use the same definitions as in the text book.

Time attributes of a task



- A task τ_j has **fixed time attributes**
  - a period of activation (sometimes) T_j
  - a (worst-case) execution time C_j
  - a relative deadline, sometimes D_j
- … **dynamic time attributes** (for each instance or occurrence)
  - an **arrival** time $a_{j,i}$
  - an **absolute deadline**, sometimes (add D_j to arrival time) $dl_{j,i}$
  - a **start** time (or beginning time) of execution $s_{j,i}$
  - a **finishing** time, also called end, completion or departure time $f_{j,i}$
- (book) Response time: $s_{j,i} - a_{j,i}$
- (book) Turnaround time: $f_{j,i} - a_{j,i}$

First-Come, First-Served (FCFS) scheduling

- Schedule the tasks in **order of their arrival**.

- Decision mode: non-preemptive
- Priority function: the earlier the arrival, the higher the priority
- Arbitration rule: random choice among processes that arrive at the same time

Tasks	execution time	arrival time
P_1	24	0
P_2	3	2
P_3	3	7

What is the **average waiting time** of the three processes **on a dual-core** platform?

First-Come, First-Served (FCFS) scheduling

- Schedule the tasks in order of their arrival.

- Decision mode: non-preemptive
- Priority function: the earlier the arrival, the higher the priority
- Arbitration rule: random choice among processes that arrive at the same time

Tasks	execution time	arrival time
P_1	24	0
P_2	3	2
P_3	3	7

- Waiting times for P_1 = 0; for P_2 = 2-2=0; for P_3 = 7-7=0, Avg. waiting time: (0+0+0)/3 = **0**

First-Come, First-Served (FCFS) scheduling

- Schedule the tasks in **order of their arrival**.

- Decision mode: non-preemptive
- Priority function: the earlier the arrival, the higher the priority
- Arbitration rule: random choice among processes that arrive at the same time

Tasks	execution time	arrival time
P_1	24	0
P_2	3	2
P_3	3	7

What is the average waiting time of the three processes **on a single-core** platform?

First-Come, First-Served (FCFS) scheduling

- Schedule the tasks in order of their arrival.

- Decision mode: non-preemptive
- Priority function: the earlier the arrival, the higher the priority
- Arbitration rule: random choice among processes that arrive at the same time

Tasks	execution time	arrival time
P_1	24	0
P_2	3	2
P_3	3	7

- The *Gantt Chart* for the schedule is:

- Waiting times for P_1 = 0; for P_2 = 24-2; for P_3 = 27-7, Avg. waiting time: (0+22+20)/3 = **14**

- **Additional exercise:** What if they arrive almost simultaneously
  - in the order P_1, P_2, P_3 ?
  - in the order P_2 , P_3 , P_1 ?
  - *Compare the two cases.*

Shortest-Job-First (SJF) scheduling

- Schedule the task with **the shortest next CPU burst (i.e., shortest execution time).**

- Decision mode: non-preemptive
- Priority function: equal to -*L* where *L* is the length of the next CPU burst
- Arbitration rule: chronological or random ordering

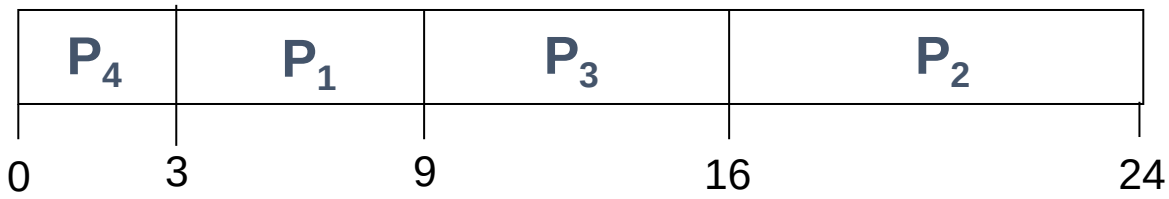
Task	execution time
P_1	6
P_2	8
P_3	7
P_4	3

What is the average waiting time of the four processes on a **single core** if they **all arrive at time 0**?

Example of SJF

Task	execution time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



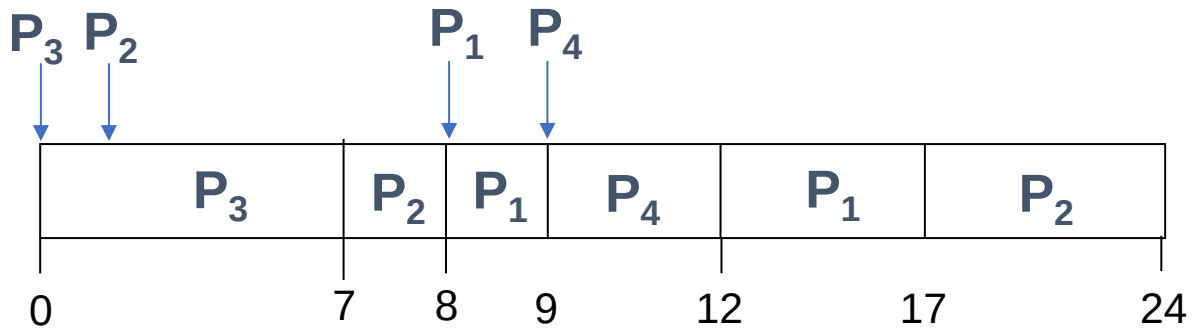
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- **SJF is optimal w.r.t. average waiting time.**
  - **It minimizes average waiting time for a given set of tasks… if you know their execution time**
- **Problem**
  - **needs prediction** of the next CPU burst length
    e.g. by using exponential weighted average of bursts in the past (see in the textbook)

Shortest-Remaining-Time-First scheduling

Preemptive version of SJF scheduling is called **Shortest-Remaining-Time-First**

Exercise: compute the **average waiting time and turnaround time** of P1, P2, P3 and P4 for the Shortest-Remaining-Time-First scheduling algorithm assuming that P3 arrives at time 0, P2 at time 1, P1 at 8 and P4 at 9

Shortest-Remaining-Time-First scheduling Solution



- **Waiting time:**
  - P3=0, P2=(7-1)+(17-8)=15, P1=0+(12-9)=3, P4=0
  - Average=(0+15+3+0)/4
- **Turnaround:**
  - P3=7, P2=24-1=23, P1=17-8=9, P4=12-9=3

Round Robin (RR) scheduling

- **Each task gets a small unit of CPU time** (***time quantum of length q***)
  - At the end of a time quantum, the task is **preempted** and is added to the end of the ready queue.

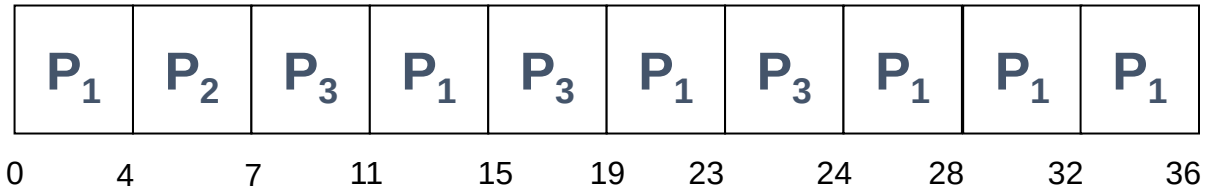
- Decision mode: if a task runs *q* continuous time units, it is *preempted*
- Priority function: the earliest it is added to the ready queue, the higher the priority
- Arbitration rule: random ordering

- **If there are *n* tasks in the ready queue**
  - **Each task gets 1/n^{th} of CPU time** (for small *q*) in chunks of at most *q* time units at once.
  - **A task waits at most (*n*-1)*q* time units until it is given CPU time again.**

Example RR schedule with time quantum = 4

Task	execution time
P_1	24
P_2	3
P_3	9

- The Gantt chart is:



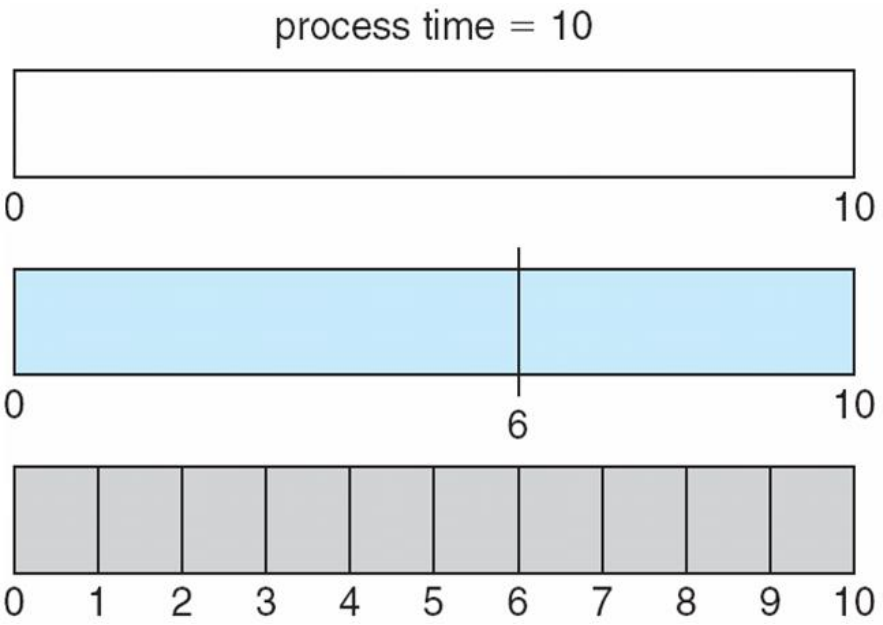
- Typically, higher average turnaround than SJF, but better *response time*.

Round Robin: Time quantum and context switch time

Performance

***q* large** ⇒ equivalent to **FCFS**

***q* small** ⇒ *q* must be large with respect to context switch time, otherwise **context switch overhead is high**.



quantum	context switches
12	0
6	1
1	9

Priority scheduling

- A priority number (integer) is associated with each task
  - explicitly assigned **by the programmer** or
  - **computed**, through a function that depends **on task properties**
    e.g., memory use, timing: duration (estimated), deadline, period

- CPU is allocated to the task with highest priority
  - **preemptive**
    - *priority_of_any_running_task ≥ priority_of_any_task_in_ready_queue*
    - *unless... there are exceptions due to sharing of resources (see slides on priority inversion)*
  - **Non-preemptive**

- Problem ≡ **Starvation** – low priority processes may never execute

  Solution ≡ **Ageing** – increase the priority of the process over time

Implemented by **sched_FIFO** and **sched_RR in Linux**

Priority scheduling: Example 1

Rate Monotonic (RM) Scheduling

In real time systems, processes are often periodic in nature…

A process with period *T* is activated every *T* time units, and its computation must complete within a relative deadline of *D* time units.

Decision mode: preemptive
Priority function: **shorter period *T* → higher priority**
Arbitration rule: chronological or random ordering

Priority scheduling: Example 2

Deadline Monotonic (DM) Scheduling

Decision mode: preemptive
Priority function: **shorter relative deadline *D*→ higher priority**
Arbitration rule: chronological or random ordering

Priority scheduling: Example 3

Earliest Deadline First (EDF)

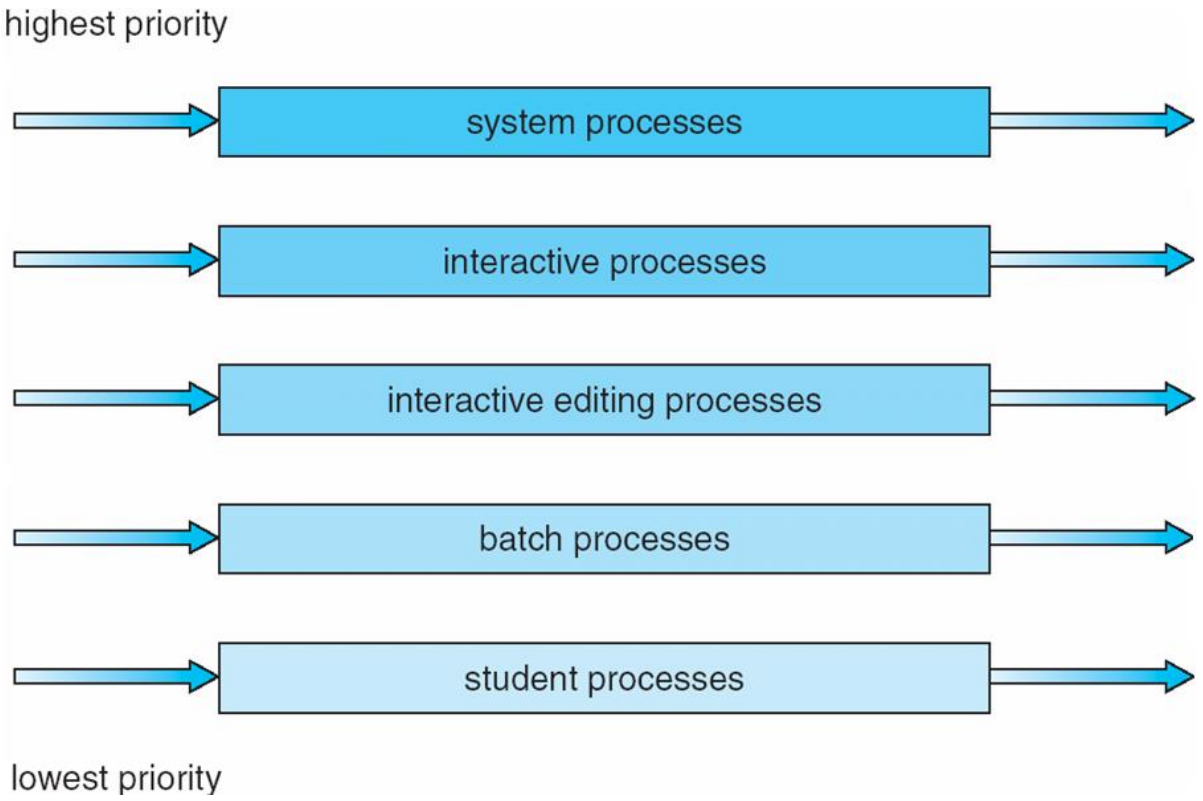
Highest priority is assigned to the process with the smallest **remaining time until its deadline**

Decision mode: preemptive
Priority function: **least time remaining until absolute deadline** → **higher priority**
Arbitration rule: chronological or random ordering

Implemented by **sched_deadline in Linux**

Multilevel queue scheduling

- Ready queue is partitioned into various sub-queues



Used in Linux to arbitrate between tasks that are assigned different scheduling policies

- Ready tasks using sched_deadline are scheduled first
- Ready tasks using sched_RR or sched_FIFO are scheduled next
- Ready tasks using the “fair scheduler” are scheduled last

Multilevel queue scheduling

- **Each queue has its own scheduling algorithm**
  - decision mode: preemptive (RR) or non-preemptive (FCFS)

- **Scheduling between the queues**
  - **Option 1:** Fixed **priority scheduling** (i.e., serve high priority queues first, and low priority queues last). → Possibility of starvation.
  - **Option 2: Time slice** - each queue gets a certain amount of CPU time which it can schedule amongst its processes; e.g.,
    - 80% to foreground processes in RR
    - 20% to background processes in FCFS

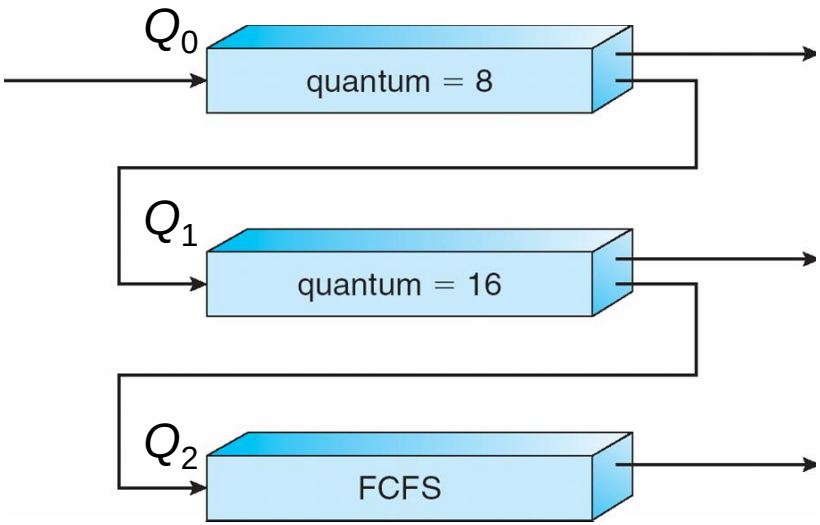
Multilevel feedback queue

- Instead of assigning a process to a fixed sub-queue, **a task can move across sub-queues**;
  - ageing can be implemented this way

- Scheduler is defined by the following parameters:
  - **number of queues**
  - **scheduling algorithms for each queue**
  - method used to determine **when** to **upgrade a task**
  - method used to determine **when** to **demote a task**
  - method used to determine **which queue a task will enter** when that task needs service

Example of multilevel feedback queue scheduling

- Three queues:
  - Q_0 – RR time quantum 8 msec
  - Q_1 – RR time quantum 16 msec
  - Q_2 – FCFS



- Scheduling
  - Job enters queue Q_0 (RR). When it gains CPU, job receives 8 msec. If it doesn't finish in 8 msec, job is preempted and moved to queue Q_1.
  - At Q_1 (RR) job receives 16 additional msec. If it still does not complete, it is preempted and moved to queue Q_2.

Summary

- Ingredients of concurrent execution: processes and threads
  - Process
    - a program in execution.
    - has a context of execution
    - owns resources
  - Thread
    - dispatchable unit of work within a process
    - shares code and data
  - Threads operate faster than processes
  - Downside: no protection against other threads (faults, memory sharing)

- Creation/deletion of process and threads
- Common processor scheduling algorithms

Friday

- **Lecture on atomicity and concurrency**
  - Traces, race conditions, critical sections, correctness of concurrent programs

- **Preparation:**
  - **2 short videos (~19 minutes)**
  - **Reference book (~25 pages)**